SPL IV: EXTENDED FORTRAN IV FOR PROCESS CONTROL

G. W. OERTER, Principal Software Engineer &
G. L. PETERMAN, Manager, Application Programming
Leeds & Northrup Company, North Wales, Pa.

Presented at the Fifth Annual Workshop on The Use of Digital Computers in Process Control, held at Louisiana State University, Baton Rouge, La. February 4 to 6, 1970

Donal

SPL IV: Extended Fortran IV for Process Control

G. W. Oerter and G. L. Peterman
Leeds and Northrup Co.

ABSTRACT

Extensions to Fortran are presented to illustrate the value of building into the language facilities for known requirements. A concept important to the processing of continuous variables is introduced: the use of modifiers which, when attached to the name of a variable, specify elementary processing on that variable. Further extension of Fortran is explored by considering the use of decision tables in program design. The advisability of building these extensions into the language is weighed against leaving their implementation to the ingenuity of the user.

INTRODUCTION

It has become common practice to provide a Fortran compiler for process control computers, preferably a ANSI (1) Fortran supplemented by system programs which can be called by standard Fortran CALL statements to provide functions and systems interfaces foreign to the Fortran language. In contrast to this approach SPL IV incorporates many extensions within the language itself, leaving only a few system interfaces to the subroutine approach. This investment in a larger compiler can be justified if the resulting system provides significant help to the user in designing, coding, debugging, and maintaining his programs while not divorcing him from the pool of viable Fortran programs and programming know-how.

It is important to make perfectly clear that SPL IV contains ANSI (1) Fortran, with the exception of complex variables. The question under discussion, then, is how best to make the necessary extensions which Fortran requires in order to be usable in a process control environment. The arguments will be pragmatic rather than theoretical, but first some background to SPL IV will be useful before considering the specifics.

*Leeds and Northrup Co., North Wales, Pa. 19454

History

In 1965 Leeds and Northrup specified a language to be used for monitoring, start-up, and shut-down of steam driven electric power generating plants. It was dubbed, "SPL", and the first system written in SPL and containing the on-line compiler was shipped in April 1967. Following that, 15 more systems were programmed principally in SPL. More than half of the systems included extensive start-up, shut-down, and plant operation guidance programs.

In these systems about 30% of the manpower on each system went into SPL programs, but these programs represented anywhere from 75% to 90% of the total system. Or, in other words, SPL cut the manpower requirements for a large system by an estimated factor of 3. In the remaining systems, about 50% of the functions were programmed in SPL IV realizing an estimated savings of 33% in programming manpower.

Unfortunately SPL came along rather late in the life of the second generation. With the arrival of the third generation it was tempting to go along with the manufacturer's on-line Fortran, nevertheless, our experience with SPL was carefully reviewed, leading to the determination to specify a new SPL, SPL IV. The IV indicates that the language is based on Fortran IV, really ANSI Fortran. The SPL part of the name originally stood for Steam Power Language; now it has been broadened to cover processes in general, but preferably ones in which the data gathering is extensive and there are many interrelated programs running at different levels (in the Schoeffler(2) sense) in the system.

SPL IV is presently implemented on third generation hardware. Its use has so far been extended from steam plants to include other applications such as cement manufacture and ore benefaction. Its functional similarity to Shell Oil's MUSIC (3) suggests that it would also be a natural in the Petroleum Industry.

SPL IV: EXTENDED FORTRAN IV FOR PROCESS CONTROL

G. W. OERTER, Principal Software Engineer &
G. L. PETERMAN, Manager, Application Programming
Leeds & Northrup Company, North Wales, Pa.

Presented at the Fifth Annual Workshop on The Use of Digital Computers in Process Control, held at Louisiana State University, Baton Rouge, La. February 4 to 6, 1970

Donal

Extensions

In going from SPL to SPL IV a policy was needed to apply to the necessary extensions of the USASI Fortran. The policy that developed was to ask, "can this function be done just as well through the subroutine structure of Fortran?". Many times the answer was "yes" as in the case of delaying a program. We could see no significant advantage to building a special DELAY statement; in fact, by using a CALL to a subroutine the function was open to desirable flexibility. In systems where the delays are used often and they are short, the subroutine can go directly to the executive. Where the delays are characteristically long, the subroutine would interface with the scheduler.

On the other hand, if building the function into the language could significantly reduce work for the user or reduce the likelihood of error, it became a strong candidate for direct implementation.

We would like to examine two such extensions in some detail. These two were selected from among several because they seem to us to be particularly significant. The first, suffixes or modifiers, is significant because it deals with a universal problem in process control: the sampling, conditioning, and checking of the plant measurements. The second, decision tables, is important because it is a facility which was extensively used in SPL and, we feel, would be widely used in all programming, if the decision table processors were built into the language system.

SUFFIXES

Before suffixes can be discussed in sufficient detail to justify them, several other characteristics of SPL IV should be explained.

System Environment

The SPL IV language and its processors (the compiler, loader, and runtime routines) are designed to interact with the total system in more ways than the mere production of object code. Figures 1 and 2 illustrate the additional system interfaces with the compiler. In addition to code produced, SPL IV outputs "messages" to the executive, which is called Central Resources Control (CRC), and the data collecting system, which is called DATAPAC, thereby automating the system interfaces so that, when an SPL IV program runs, all its resources are co-resident and current. DATAPAC will be referred to as the Data Acquisition program in this paper although its functions are broader than that.

CRC, the executive, provides a multi-programming environment characterized by classes of priorities rather than individual priorities for each task. A request for a task in the adjacent class will not preempt the running program, but a request two classes away will cause preemption provided that the running program has been in control for one time slice, 100 milliseconds in steam plant applications. This priority handling scheme tends to minimize the swapping of core images to and from secondary memory while still insuring fast response to high priority functions.

CRC, processes a request by reference to a central table which describes all entities in the system. The SPL IV loader registers each SPL IV program as an entity in this table at load time. The loader also registers the symbolic name of the program with the background processor, LNMONITOR,so that other programs and the operator can then refer to the newly loaded program by name.

DATAPAC, the data gathering sub-system, is a collection of table driven tasks which together perform the data sampling, the man-machine interface with the plant operator, and all the functions the operator can directly ask for, such as trends, special logs, point summaries and so forth. DATAPAC does considerable processing of individual data under the direction of the suffixes to be described later. Since DATAPAC runs continuously, in parallel with SPL IV programs and maintains precise sampling intervals, it is well suited to handle time-dependent functions such as integration and differentiation.

SPL IV interfaces with DATAPAC both at compile time and at runtime. These interfaces are implemented by a general structure in order to retain flexibility in defining their meaning for particular applications. These interfaces will be described in more detail when suffixes are discussed below.

System-wide Names

The concept of system-wide names or identifiers is very simple. They can be compared to implicit typing in Fortran where variables whose names begin with any of the letters I,J, K,L,M,N, are assumed to be integer while variables whose names begin with any other letter are assumed to be floating point. In system-wide names the idea is carried further, also the names are valid not only in the compiler, but also in the man-machine interface. Since these names refer to very real things, there is no reason why they should not be as unique and as permanent as the names of people. The

motivations for system-wide names are: the standardization of documentation, simplicity in the man-machine interface, and simpler system programs. System-wide names are used for all variables in DATAPAC and for all variables which cross the DATAPAC/SPL IV interface in either direction.

These system-wide names must have a general format to distinguish them from local SPL IV variables. This general format is PPdddSSS, where PP is a two-character prefix, ddd is a three-digit number, and the S's are optional suffixes. As in all Fortran identifiers, the first "P" must be a letter of the alphabet.

The set of prefixes is determined by the particular application. It may follow the ISA standard S5.1, if that is desirable; for instance:

AT - Analysis Transmitter
FT - Flow Transmitter
TC - Temperature Control
LC - Level Control

The numbers, ddd, are arbitrary and their range is determined for each application. The suffixes may also be chosen for a particular application. The following are a few typical suffixes:

A - Two minute average
C - Running average
M - Hourly maximum
I - One hour integration
N - Hourly minimum
R - Rate of change
S - Smoothing

TC01OM is the system-wide name of the hourly maximum of the variable derived from the thermocouple input number ten. The suffixes when present are applied in sequence reading from left to right, thus, a common variable is FT001SA, it is first sampled, then smoothed, then averaged. The suffixes will be discussed further below.

Cumulative Records

Another important characteristic of SPL IV is that it assumes that the user is not creating an isolated program, but that the program being compiled may be related in some way to other programs in the system. To simplify these relationships the SPL IV system keeps certain records over the life of the system. The user has control over this record keeping by means of the CLASS statement. To get the cumulative records he must declare near the beginning of his program:

```
CLASS       SPL
```

The default case (no CLASS statement) is Fortran, causing the program to be treated as standard Fortran, but still permitting the use of decision tables.

In Fortran, SUBROUTINE and FUNCTION subprograms are given names so that they may be called by other programs. In SPL IV main programs are also given names such as:

```
PROGRAM          MARDIGRAS
```

This declaration is complemented by the statement:

```
GO TO MARDIGRAS
```

Thus it becomes very simple to link programs together in a chain to accomplish segmentation. The SPL IV loader registers a program by name with the system at load time. From that point on the loaded program may be activated by any other program in the system or by the operator through the LNMONITOR.

A second type of system build-up takes place at the variable level. Data is passed from program to program through communication areas of which there are three types.

1) DPCOMAREA's are used for variables generated by DATAPAC for SPL IV. So these are strictly input areas representing the continuous variables of the plant. The number and size of these areas is flexible.

2) COMAREAS are like DPCOMAREAS but are used for communicating data values between SPL IV programs.

Both of these types of communication areas reside on secondary memory. Their memory allocation is independent of the programs which reference them.

3) SYSCOM is a core resident communication area which combines the characteristics of both the other area types. There can be only one SYSCOM.

For all these areas SPL IV keeps permanent symbol tables; thus the areas can be built up variable-by-variable over the life of the system. The allocation of each variable is by name rather than by position. As a result, the programmer need declare only the variables he wishes to use. If any of his variables are new to the COMAREA they will be added, otherwise they will be referenced according to their first declaration. It is possible, then, to know and to control

where the variables are located if that is desirable in order to equivalence them to an array.

Suffixes

Since the concept of suffixes is unusual if not unique, it will be good to try to get an intuitive idea of them first. Fortran is called a language, so it should be possible to identify parts of speech in Fortran statements. Things like AIXPS, YOUFS, etc. are clearly proper nouns. Things like GO TO, READ, etc., are clearly transitive verbs; so also are the mathematical and logical operators because they really say, for instance, "add AIXPS to YOUFS". Other parts of speech are hard to find in Fortran, although newer languages increasely exhibit the tendency to develop in the same direction as natural language. The structures in PL/I, for instance, can be considered common nouns, rather than proper nouns, thus it is possible to have a chair or a person described in a program by means of a structure.

What about adjectives? Adjectives are very important to meaning in natural language. Consider water for instance. Suppose I offered you a drink of water. You might want to know more than the fact that its basic chemical make up is H_2O. Is it clean, hot, cold, cool, lukewarm, pure, poisoned? An analogy can be drawn to continuous variables. It is often desirable to have the variable smoothed; averaged; integrated, differentiated, or otherwise transformed from its raw sampled state before computing with it.

Both sets of adjectives given above imply some sort of processing. To be hot the water must have been heated and so forth. But how could this be true in a programming system, that is, how could an adjective imply pre-processing? In dealing with continuous variables, desirable transformations of those variables should be available in the system without explicit programming on the users part. This is possible only if processors, which can perform the various conditioning, already exist in the system as independent tasks and the high level language can invoke these processors. Subroutine implementation would be extremely awkward and difficult because the system is dealing with continuous variables. Suppose for instance, that the average of all the samples between each execution of the high level program is what is desired for a given variable. Either the high level program must run at each sampling interval and do its own averaging or else a separate task must be created, its outputs stored in a pre-arranged place, and an interface subroutine pick them up when called by the high level program. The latter is what is done in SPL IV in an automated way. The Data Acquisition sub-system contains the separate tasks need. The communications areas provide the pre-arranged storage, and the system-wide names make the intent of the SPL IV user explicit to the system.

How Suffix Processing is Invoked

The following example will illustrate everything the user must do in order to take advantage of the parallel processing provided by the Data Acquisition Sub-system. Suppose that we want to make use of an enthalpy which is a function of temperature and pressure. We can examine the total programming requirements step-by-step.

1) We need two inputs, a pressure and a temperature so they are obtained by the following declaration written in the SPL IV program.

```
DPCOMAREA/MYOWNIN/PT01OSA, TC01O
```

The system effect of the above statement will be to get the communication area MYOWNIN whenever this program runs. Meanwhile DATAPAC will supply fresh values of the pressure and the temperature. The pressure sample will first be smoothed and then averaged before being placed in the communication area MYOWNIN..

2) The writing of the calculation is straightforward.

```
ENTVAP = ENT (TC01O, PT01OSA)
```

ENT is the usual Fortran library function call; TC01O and PT01OSA are the arguments.

3) Suppose, however, that we want this enthalphy to be available to the operator for display, trending, whatever, then we write.

```
CV001 = ENT (T01O, PT01OSA)
```

The CV001 is recognized as a calculated value because it has a system-wide name. This is the term for values passed back to DATAPAC from SPL IV. In addition to storing the value locally, the SPL IV system will pass the value, the name, and the time calculated to the Data Acquisition sub-system. There it will be stored in a file so that it can be retrieved for any DATAPAC purpose. In this way values can be "picked off" and made available to the man-machine interface as they are generated.

4) Finally, we may want to specify some things about the DATAPAC processing of these variables. The following

statements would set the sampling frequencies and give an English identification to each of the variables.

```
DP SPEC
TC010:
PSCAN = 6,
ID = 'TEMP OF SATURATED VAPOR'
PT010:
PSCAN = 3,
ID = 'PRESSURE OF SATURATED VAPOR'
END SPEC
```

There are several interesting things to note about the above code. Since the Data Acquisition programs are table driven, the object of the code is to set certain values in the tables which drive the processing of the variables in question. "DP SPEC" warns the compiler that something peculiar follows. The "TR010:" indicates that the following substitutions apply to the processing table or TR010 rather than to compiled code. The compiler, therefore, outputs the substitutions to DATAPAC rather than compiling them as internal code. The "variables" on the left, PSCAN, ID, are really reserved words (within a DP-SPEC) from a set of 64 which describe to DATAPAC processing to be performed on plant variables. The compiler contains a table which defines the type of value which should appear on the right of the equal sign. The DATAPAC system accepts the compiler output and stores the values in the proper fields of the processor tables. A limited number of these fields can also be set dynamically at execute time through the VARSPEC statement. When done dynamically, the values on the right can be determined by variables or expressions in the manner of standard arithmetic substitution statements.

For those familiar with PL/I the mechanism can be explained as follows. TC010 is the name of a variable whereas, TC010: is the name of a structure. The reserved variables, PSCAN, ID, HI, LOW, etc. are elements of the structure TC010:. They are not homogenous, rather each one has its own set of attributes; in many instances the element occupies less than a full word, often a single bit.

The programming savings noted earlier under the history of SPL are achieved largely through the above mechanisms which significantly reduce the clerical work of the user.

DECISION TABLES

The main impetus behind decision tables has been their value as clear documentation. Everyone interested in the programming problem should read Fisher's paper [4] and do his exercises to see what decision tables are all about. The question arises then, as to why they are not more widely used; why are they not exploited by the users of procedural languages? Our experience has been that SPL users exploit the decision table. The very first SPL program delivered has 340 decision tables in it by actual count. We have theorized that the decision tables are readily accepted in SPL because they are an integral part of the language. No special treatment such as hand translation, special passes, or pre-passes of the compiler are required. They are executed in line with the other code and the compiler generates the tables in a compact form so that they use memory sparingly.

In this paper we would like to claim another virtue for decision tables. Because of the clarity of the logic as expressed in the decision tables, they make a good tool to help in the design of the procedure. Thus they contribute to easier programming not only after the fact, but also before the fact. To attempt to illustrate this we will discuss a simple problem which has been solved and documented using a method called "Control Logic Diagrams" (5). In this way we hope to illustrate the simplicity and thoroughness of the decision table approach.

Figure 3, pictures a centrifugal pump connected to a valve and indicates the sensors and actuators required with this combination. We do not know the exact mental process used to arrive at the control logic diagram shown in the referenced article, but we will develop the required decision table systematically and independently.

First, we must know the functional requirements. These were stated as follows; "the pump must be started before the valve is opened, and the valve must be closed before the pump is stopped. If pump, motor, or coupling fails, the valve is to be closed at once". Comparing this statement with the instrumentation shown in figure one, we can see that the starting and stopping of the pump will be determined by the states of three digital inputs, the start button (DI001), a valve closed (DI004), and valve open (DI005).

From the above we can now construct a preliminary decision table showing all the combinations of the variables.

	RULES							
	1	2	3	4	5	6	7	8
START BUTTON ON	Y	Y	Y	Y	N	N	N	N
VALVE OPEN	Y	Y	N	N	Y	Y	N	N
VALVE CLOSED	Y	N	Y	N	Y	N	Y	N
START PUMP	N	N	Y	N	N	N	N	N
STOP PUMP	N	N	N	N	N	N	Y	N

Examining each rule, we can make several observations about the process. Rules 1 and 5 represent error conditions; this state of the limit switches should never occur. It can occur, however, through a faulty switch so we might have these rules in the final table and show an error action. Rules 4 and 8 show that the valve is in travel; again we might want to recognize this state and time the travel by means of a program delay and alarm if the state does not change in reasonable time. Next we see that rule 2 represents the normal pumping condition, hence no action is required. Rule 3 represents the proper start conditions; the valve is closed and the operator has pressed the start button. Rule 6 shows that the operator is calling for the pump to stop, but the valve is still open, therefore the pump should not be tripped yet. Rule 7 gives the proper conditions for tripping the pump; the valve and the operator are now in agreement.

We can now reduce the table to the following three rules. The third rule, where I is read "don't care", is usually called the "else" rule because it includes the remaining possibilities. It is important to realize that the left-most true rule is the one which is executed so rule three, which is always true, will be executed only if neither rule one nor rule two is true.

	RULES		
	1	2	3
START BUTTON ON	Y	N	I
VALVE OPEN	N	N	I
VALVE CLOSED	Y	Y	I
START PUMP	Y	N	N
STOP PUMP	N	Y	N

Actually the original unreduced table would take no more memory or time in SPL IV than the reduced table, so it becomes a matter of documentation preference which table is used.

The above table does not agree exactly with the published control logic diagram. To represent the logic diagram exactly we would have to construct the following table.

	RULES		
	1	2	3
START BUTTON ON	Y	N	I
VALVE OPEN	I	N	I
VALVE CLOSED	I	Y	I
START PUMP	Y	N	N
STOP PUMP	N	Y	N

We can see that this table, as well as the diagram, unconditionally starts the pump when the button is pushed. We can also see from the earlier development of the complete table that rule 1 could violate an operating principle set down at the beginning - "the pump must be started before the valve is opened". This condition should never occur in normal operation, however, the version of the decision table which we independently developed guards against the possibility, should it ever occur.

This example has been carried out using what are commonly called limited entry decision tables. Limited entry tables are easy to follow, easy to translate, and easy to change on-line; however, SPL IV also offers extended entry tables. The extended entries have the advantage of producing more compact tables, which also implies less writing for the user. Also the extended entry tables are often easier to conceive when dealing with continuous variables rather than the discrete variables in the examples.

The following is an example of an extended entry.

	RULES				
	1	2	3	4	5

IF(TCO1OA.LE.#): ALPHA, BETA, GAMMA, DELTA, OMEGA

The # sign can be considered a blank which must be filled in by the entries on the right. If the system were confined to limited entries, the above statement would have to be expressed as five statements as follows:

	RULES				
	1	2	3	4	5
IF (TCO1OA.LE.ALPHA):	Y,	I,	I,	I,	I
IF (TCO1A.LE.BETA):	I,	Y,	I,	I,	I
IF (TCO1OA.LE.GAMMA):	I,	I,	Y,	I,	I
IF (TCO1OA.LE.DELTA):	I,	I,	I,	Y,	I
IF (TCO1OA.LE.OMEGA):	I,	I,	I,	I,	Y

The advantage gained by including decision tables in the language is that this desirable logic structure is made easily accessible to the user. Other advantages accruing from special implementation have been discussed before (6), the

most dramatic one being the amount of storage saved by using the decision tables rather than branches in Fortran.

Other Extensions to Fortran

In order to round out the discussion of built-in extensions to Fortran several more SPL IV extensions should be mentioned.

The first of these is a debugging aid, the X compilation mode. This makes it possible for the user to incorporate extra statements in his program for debugging purposes. These extra statements are marked with an X in column one. In the X mode they are compiled, in the normal mode they are not compiled, so removing the debugging steps from the final program. The X mode is particularly useful in paper-tape oriented systems where editing of source code is more strenuous than in card-oriented systems.

A second extension is the packing of boolean variables sixteen to a word. These variables are core resident and have system-wide names, making them instantly available to all processors, programs and people in the system. The LOGICAL class of variables are distinguished from booleans and are treated a la Fortran for compatibility with existing programs.

There are additional system interfaces. For instance the declaration "END" results in compiled code, namely an exit to CRC. This guarantees that all SPL IV programs will exit gracefully whether or not they contain an EXIT or STOP statement. In addition to linking program segments together by means of the GO TO PROGRAM statement, the user can also state, CALL SUBPROGRAM. This differs from a subroutine call only in that the SUBPROGRAM will not be allocated to be co-resident with the main program. Thus a SUBPROGRAM call goes through the executive, CRC, whereas a SUBROUTINE call is strictly internal to the running program. By this mechanism large sub-routines can be kept in secondary memory until called.

A pair of statements, STEP and CHECKPOINT, were kept in SPL IV to simplify the writing of start-up and shut-down programs by providing a simple structure for isolating ambiguous events. These functions could be implemented using subroutine calls but they are used so frequently in start-up and shut-down programs that they appear in the language because of the convenience afforded by the shortened form. Also, the STEP statement often appears in the action stub of decision tables where the shortened form is desirable to maintain the table format readability.

One more extension should be considered because it reflects the industrial environment in which the SPL IV language is expected to operate. A three-bit quality field is included in every real variable, so the value of a variable consists of three parts instead of the usual two parts. A typical floating point variable consists of a mantissa and an exponent; SPL IV floating point variable consist of a mantissa; quality, and exponent. The quality field receives its initial value from the Data Acquisition sub-system, DATAPAC. The following table gives the qualities assigned by DATAPAC.

Code	Printed Character	Interpretation
ø	(blank)	Good
1	H	Good (high operating limit)
2	L	Good (low operating limit)
3	D	Doubtful, suspect, imperfect
4	S	Substitute (manual)
5	U	Unreasonable (high transducer limit)
6	V	Unreasonable (low transducer limit)
7	*	Bad (beyond ADC range)

The quality field is preserved and propagated by the SPL IV floating point arithmetic. Propagation follows the rule that the quality of the result is equal to the quality of the worst input to the calculation. The quality field can, of course, be set, tested, and printed with the value of the variable. It is often used as a switch to determine alternate methods of calculation. The SPL IV program sometimes sets the quality as a function of other variables. For instance, in start-up, a measurement may not be meaningful at certain stages of the start-up so the SPL IV program is written to set the quality to "doubtful" or "bad".

The quality field is an interesting extension in that it does not have a direct effect on the language or the compiler. It is rather a run-time phenomena. Special statements could have been included to handle quality, but the Fortran language already had sufficiently powerful mechanisms for dealing with the quality field provided that the run time package had the facility to preserve and propagate it.

Summary

We have tried to show that considerable programming manpower can be saved by orienting a language to the system in which its object programs are to run. This orientation requires

an expansion of the compiler and loader so that they will do clerical jobs ordinarily left to the user. We have also tried to illustrate the benefits of including new structures which break out of the narrow programming mold of loops, arrays, and procedures. When compared with natural languages, programming languages show that they have a long way to go before they duplicate the efficiency of the basic parts of speech found in natural languages. This paper has suggested the usefulness of adjectives and common nouns in programming language.

The principle behind SPL IV is the familiar one of tooling. Without the modern concept of "tooling up" manufacturing would still be in the age of craftsmen. Likewise programming will not get out of the age of craftsmen until more comprehensive tools are devised.

References

1. W. P. Heising, et al. USA Standard Fortran X3.9-1966. American National Standards Institute (Formerly USASI). 1966

2. James D. Schoeffler, "Process Control Software", Datamation February, 1966.

3. W. R. Biles, et. al. "MUSIC", Series of four papers presented at 64th AICHE meeting, New Orleans, La., March, 1968.

4. D. L. Fisher, "Data Documentation and Decision Tables", Communications of the ACM, Vol. 9, page 26, (1966).

5. G. Turner, "On-Line Programs from Control Logic Diagrams", Control Engineering, Vol. 15, No. 9, September, 1968.

6. G. W. Oerter, "A New Implementation of Decision Tables for a Process Control Language, IEEE Transactions on Industrial Electronics and Control Instrumentation, Vol. IECI-15, No. 2, December, 1968.

TYPICAL ON-LINE COMPILER

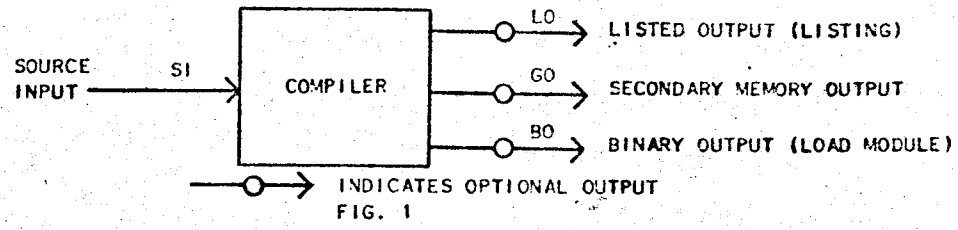

FIG. 1

SPL IV ON-LINE COMPILER

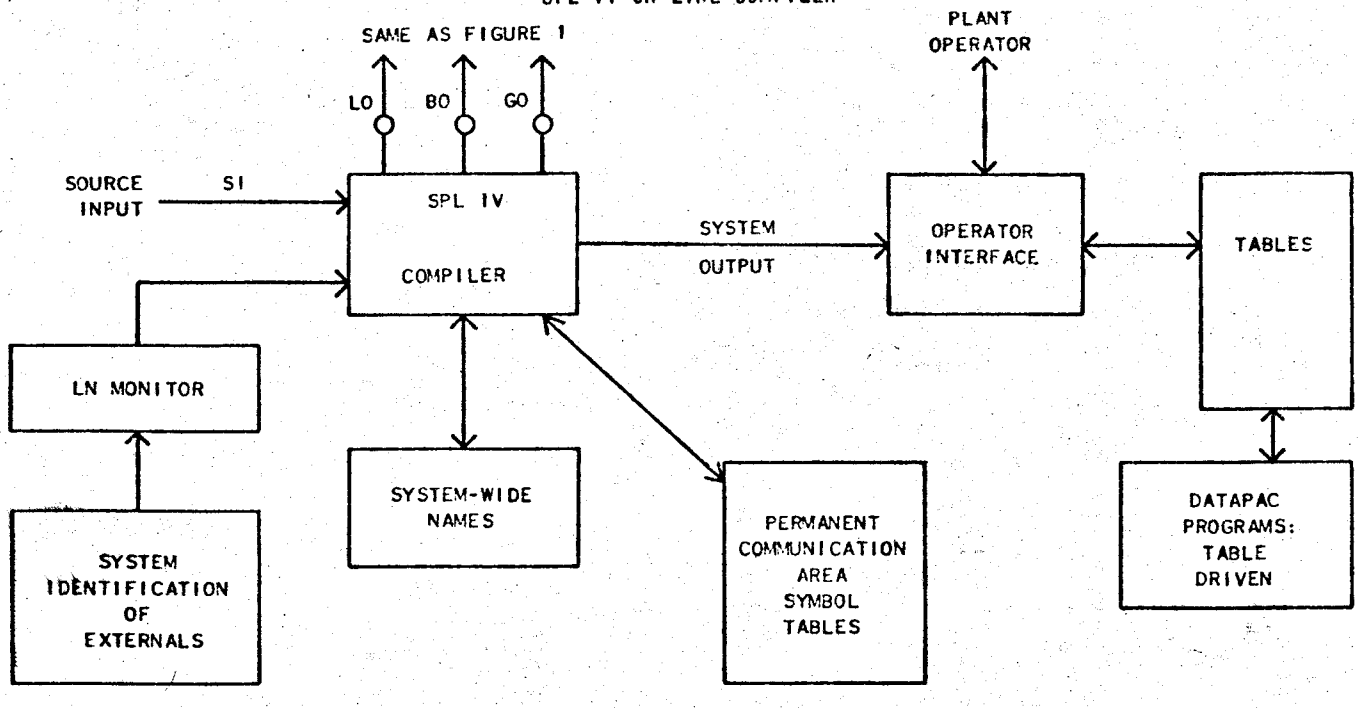

FIG. 2

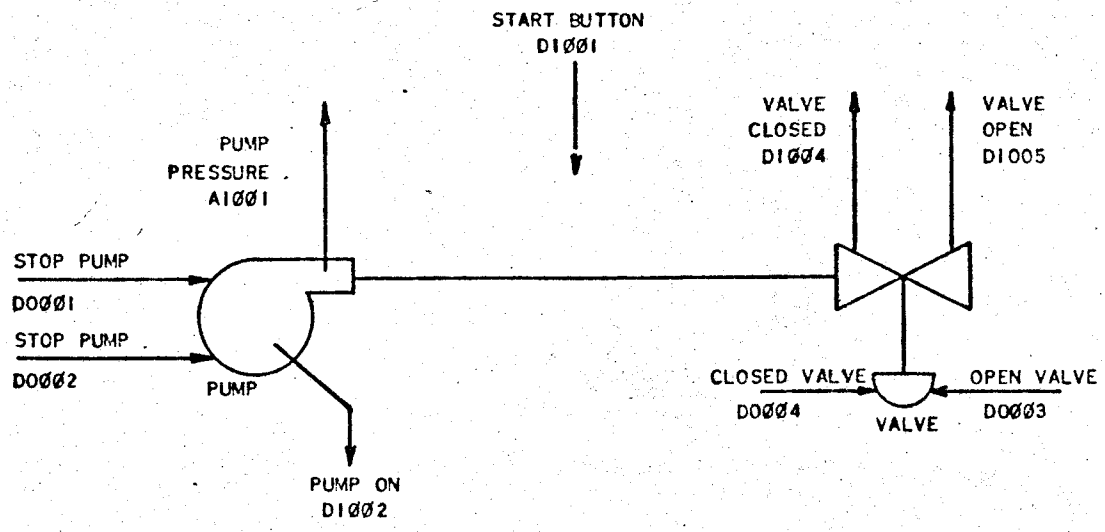

FIG. 3